36 Casting and Coercion of type

Types are extremely specific. An integer scalar type is not an integer array type.

However, Visual APL has a set of rules for coercing between types. This allows you to cast one type as another if possible.

For instance:

```
      a = 10.3
      b = (int) a
      b
10
```

We have coerced a double to an Int32 integer. We could also cast an integer to a double:

```
      a = (double) b
      a
10
      Ÿdr a
643
      a.GetType()
System.Double
```

It is not possible to create explicitly typed variables in the session. If you want to try this with explicit typing of variables you will need to do that within a function.

All objects have the method `GetType` which returns the current type of the object. This works similar to `Ÿdr`, but with `GetType` it is the responsibility of the object to return its type. With `Ÿdr` it is the responsibility of a separate function, `Ÿdr` in this case, to determine the type of a variable.

It is also possible to cast arrays, for instance:

```
      c = (Double[]) 1 2 3 4 5
      c.GetType()
System.Double[]
```

Since Visual APL is fundamentally designed to include arrays, you can also simply type:

```
      c = (Double) 1 2 3 4 5
      c.GetType()
System.Double[]
```

While the first instance explicitly states that a vector is to be returned, the second only specifies cast, but does not explicitly indicate that an array will result.

It is also possible to cast a string to integer:

```
a = (int) "abc"
      a
97 98 99
```

However, if you cast the integer array to string, you get the string display of the integers, for instance:

```
a  = (string) 97 98 99
      a
97 98 99
      a.GetType()
System.String
```

This behavior is compatible with other .Net languages, as casting something to string returns the string representation of the object. All object also have a ToString method which is the method the object uses to display itself. However, the way an object chooses to display itself may be somewhat surprising.

For instance:

```
      a = (int) 1 2 3 4 5
      a.GetType()
System.Int32[]
      a.ToString()
System.Int32[]
```

Even though we might have expected to see the numbers 1 through 5 displayed, the Int32 object chooses to display only its data type.   Again, each object chooses how to display itself using the ToString method.

If you had wanted to convert the 97 98 99 back to abc then you could have done two casts:

```
      b = (string) (char) 97 98 99
      b
abc
```

By first casting to characters, we then have the string method on a character array, which then displays the abc result.

We have also included Ÿucs to convert integers to unicode and unicode to characters:

```
      Ÿucs "abc"
97 98 99
      Ÿucs 97 98 99
abc
```

Because all text data in Visual APL is based on unicode, you can by default display any unicode character, for instance:

```
      Ÿucs 'Ā'
8714
```

The APL epsilon is located at character position 8714 in unicode.

When casting to a variable that has been typed, there are several issues to note. First if we type a variable as integer array, this must be done in a function, not in the session, we can use:

```
      Int32[] a = ŷ10
      Ÿùa
0 1 2 3 4 5 6 7 8 9 10
```

If we were to create a as an integer singleton, we could do the following:

```
      Int32 a = ŷ10
      Ÿùa
0
```

In this case "a" can only be an integer singleton, and the first element of the integer array created by •IO is placed in the strong typed variable "a".

Many types will cast to other types, however, it is not possible to cast disparate types to each other, for instance:

```
      a = ŷ10
      b = (Hashtable) a
Invalid cast: System.Collections.Hashtable
```

The error thrown indicates that this is an invalid cast.

Casting is particularly important when calling methods from other assemblies. For instance, with our IndexOf example, we can use castings as follows:

```
      a = (string) 10Ź'hello'
      b = 10.0
      c = 2Ź"el"
      a.IndexOf((string) c, (int) b)
1
```

In this case we took data which was not the types expected by the external IndexOf method on the string type and cast the variable c and b to the correct types for the IndexOf method.

While the automatic type selection process will in most cases choose the correct method, by casting we guarantee that the data is coerced to the types desired for the method overload we want.